МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №1

по дисциплине

‘Низкоуровневое программирование’

Вариант №1

Выполнил:
Студент группы P33312
Соболев Иван
Александрович
Преподаватель:
Кореньков Юрий
Дмитриевич

Санкт-Петербург, 2023

Цель:

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное
обновление) информации общим объёмом от 10GB соответствующего варианту вида.

Задачи:

1. Спроектировать структуры данных для представления информации в оперативной памяти

a. Для порции данных, состоящий из элементов определённого рода (см форму данных), поддержать тривиальные значения по меньшей мере следующих типов: цетырёхбайтовые целые числа и числа с плавающей точкой, текстовые строки произвольной длины, булевские значения

b. Для информации о запросе

2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним:

a. Операции над схемой данных (создание и удаление элементов схемы)

b. Базовые операции над элементами данных в соответствии с текущим состоянием схемы (над узлами или записями заданного вида)

i. Вставка элемента данных

ii. Перечисление элементов данных

iii. Обновление элемента данных

iv. Удаление элемента данных

3. Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс со
следующими операциями над файлом данных:

a. Добавление, удаление и получение информации об элементах схемы данных, размещаемых в файле данных, на уровне, соответствующем виду узлов или записей

b. Добавление нового элемента данных определённого вида

c. Выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных (по свойствам/полями/атрибутам и логическим связям соответственно)

d. Обновление элементов данных, соответствующих заданным условиям

e. Удаление элементов данных, соответствующих заданным условиям

4. Реализовать тестовую программу для демонстрации работоспособности решения

a. Параметры для всех операций задаются посредством формирования соответствующих структур данных

b. Показать, что при выполнении операций, результат выполнения которых не отражает отношения между элементами данных, потребление оперативной памяти стремится к O(1) независимо от общего объёма фактического затрагиваемых данных

c. Показать, что операция вставки выполняется за O(1) независимо от размера данных, представленных в файле

d. Показать, что операция выборки без учёта отношений (но с опциональными условиями) выполняется за O(n), где n – количество представленных элементов данных выбираемого вида

e. Показать, что операции обновления и удаления элемента данных выполняются не более чем за O(n*m) > t O(n+m), где n – количество представленных элементов данных обрабатываемого вида, m – количество фактически затронутых элементов данных

f. Показать, что размер файла данных всегда пропорционален количеству фактически размещённых элементов данных

g. Показать работоспособность решения под управлением ОС семейств Windows и *NIX

5. Результаты тестирования по п.4 представить в составе отчёта, при этом:

a. В части 3 привести описание структур данных, разработанных в соответствии с п.1

b. В части 4 описать решение, реализованное в соответствии с пп.2-3

c. В часть 5 включить графики на основе тестов, демонстрирующие амортизированные показатели ресурсоёмкости по п. 4

Исходный код проекта:

Ivanio1/LLP-1 (github.com)

Описание работы:

Программа представляет собой консольное приложение, позволяющее хранить, редактировать и просматривать данные в формате документного дерева.

Программа состоит из следующих модулей:

- File_managing – модуль для работы с файлом (Открытие, закрытие, чтение, запись, инициализация).
- Data_managing – модуль для работы с основными структурами данных (Чтение из файла, запись в файл).
- Crud – основной модуль, в котором описаны базовые операции с данными и публичный интерфейс для работы с этими операциями.
- String_managing – вспомогательный модуль для работы со строками.
- Test – модуль, в котором содержится тестовая программа.
- User_interface – вспомогательный модуль для удобного создания структур данных и работы с ними из консоли.

Примеры работы программы:

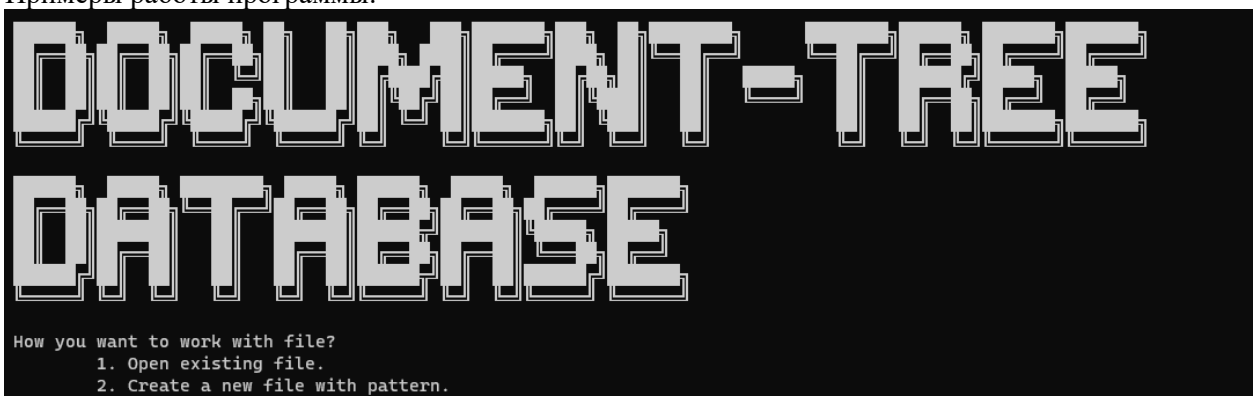


Рисунок 1-Выбор режима работы с файлом

```
How you want to work with file?
        1. Open existing file.
        2. Create a new file with pattern.
>>>2

Enter file name> file.txt


Please, initialize the pattern of your data.
How many fields in pattern? - 3
FIELD 1
Field name: number
Field type (0-Boolean, 1-Integer, 2-Float, 3-String): 1
FIELD 2
Field name: name
Field type (0-Boolean, 1-Integer, 2-Float, 3-String): 3
FIELD 3
Field name: flag
Field type (0-Boolean, 1-Integer, 2-Float, 3-String): 0
Test_mode - 1; Regular_mode - 0:0
Success!
'man' for available commands.
```

Рисунок 2-Заполнение паттерна

```
How you want to work with file?
        1. Open existing file.
        2. Create a new file with pattern.
>>>1

Enter file name> file.txt
Test_mode - 1; Regular_mode - 0:0
Success!
'man' for available commands.
```

Рисунок 3-Работа с существующим файлом

```
'man' for available commands.
man
Available commands:
add
update
remove
find
print
exit
```

Рисунок 4-Вывод доступных команд

```
add
Enter fields of new cortege
number              :333
name                :ttt
flag                :1
```

Рисунок 5-Добавление элемента

```
print
|       CORTEGE   0       |
number              111
name                www
flag                true
|~~~~~~~~~~~~~~~~~~~~~~~~~|

|       CORTEGE   1       |
number              222
name                rrr
flag                false
|~~~~~~~~~~~~~~~~~~~~~~~~~|

|       CORTEGE   2       |
number              333
name                ttt
flag                true
|~~~~~~~~~~~~~~~~~~~~~~~~~|
```

Рисунок 6-Вывод всех элементов

```
update
Enter id
Id                  :2
Choose field
0. number
1. name
2. flag
Field: 1
Enter new value:
Ivan
print
|       CORTEGE   0       |
number              111
name                www
flag                true
|~~~~~~~~~~~~~~~~~~~~~~~~~|

|       CORTEGE   1       |
number              222
name                rrr
flag                false
|~~~~~~~~~~~~~~~~~~~~~~~~~|

|       CORTEGE   2       |
number              333
name                Ivan
flag                true
|~~~~~~~~~~~~~~~~~~~~~~~~~|
```

Рисунок 7-Обновление элемента

```
remove
Enter id
Id                  :2
print
|       CORTEGE   0       |
number              111
name                www
flag                true
|~~~~~~~~~~~~~~~~~~~~~~~~~|

|       CORTEGE   1       |
number              222
name                rrr
flag                false
|~~~~~~~~~~~~~~~~~~~~~~~~~|
```

Рисунок 8-Удаление элемента

```
find
Choose type of find
1. Find by id
2. Find by parent
3. Find by field
Type                :1
Enter id
Id                  :1
number              : 222
name                : rrr
flag                : 0
```

Рисунок 9-Поиск по id

```
find
Choose type of find
1. Find by id
2. Find by parent
3. Find by field
Type                :3
Choose field
0. number
1. name
2. flag
Field: 1
Enter expression to equaling:rrr
--- CORTEGE   0 ---
number              : 222
name                : rrr
flag                : 0
```

Рисунок 10-Поиск по полю

```
add
Enter fields of new cortege
number              :444
name                :www
flag                :1
Write parent id     :1
find
Choose type of find
1. Find by id
2. Find by parent
3. Find by field
Type                :2
Enter parent id
Id                  :1
--- CORTEGE   0 ---
number              : 444
name                : www
flag                : 1
```

Рисунок 11-Поиск по родителю

```
exit
Bye bye! See you again (◕‿◕)
ivan@LAPTOP-8ANVIHN8:~/LLP-1$
```

Рисунок 12-Завершение работы

Аспекты реализации:

Внутреннее описание созданной программы:
Для обеспечения хранения всей информации в одном файле было решено разделить его на две части – метаданные и основные данные.

Секция с метаданными хранит в себе – информацию о типах и названии полей, массив индексов и текущий указатель кортежа (вспомогательный элемент, который нужен для отслеживания позиции последнего модифицированного элемента). Массив индексов (идентификаторов) для каждого кортежа содержит отступ до него от начала файла, что позволяет за O(1) находить любой элемент по id.

```c
/**
 * Container storing file parameters
 * cur_id          - sequence for tracking the current id
 * pattern_size    - number of fields in the vertex template
 */
struct tree_subheader {
    uint64_t cur_id;
    uint64_t pattern_size;
};


/**
 * Container for the pattern of key
 */
#pragma pack(push, 4)
struct key_header {
    uint32_t size;
    uint32_t type;
};
struct key {
    struct key_header *header;
    char *key_value;
};
#pragma pack(pop)



/**
 * Full file header
 */
struct tree_header {
    struct tree_subheader *subheader;
    struct key **pattern;
    uint64_t *id_sequence;
};
```

Секция с данными хранит элементы с их значениями. При проектировании и первых попытках реализации возникла проблема размера кортежа. Хотелось хранить кортежи одинаковых размеров, чтобы в случае удаления можно было на место «дырки» просто переместить новый кортеж. С типами int, bool и float было все ясно, а вот со строковым типом было непонятно, как разместить строки различной длины, при этом имея фиксированный размер кортежа. Для решения данной проблемы были введены два типа кортежей – обычный и строковый. Обычный кортеж хранит отступ до родительского и непосредственно массив данных кортежа, в котором все типы данных, кроме строкового, лежат в явном виде. Строковый тип хранится в виде ссылки на строковый кортеж. Сам же строковый кортеж хранится в виде двусвязного списка. Пример в коде:

```c
/**
 * Structure for header of object depending on its type
 */
union cortege_header {
    struct {
        uint64_t parent;
        uint64_t alloc;
    };
    struct {
        uint64_t prev;
        uint64_t next;
    };
};

/**
 * Main object in file
 */
struct cortege {
    union cortege_header header;
    uint64_t *data;
};
```

```c
for (size_t i = 0; i < size; i++) {
    if (types[i] == STRING) {
        //Добавляем ссылки на строковые кортежи
        insert_string_cortege(file, string: (char *) fields[i], cortege_size: get_cortege_size( pattern_size: size), par_pos, str_pos: link);
        new_cortege->data[i] = *link;
    } else {
        new_cortege->data[i] = (uint64_t) fields[i];
    }
}
```

Данное решение помогло справиться с проблемой разно размерных кортежей.

Особенности алгоритмов с примерами кода:

Для того, чтобы различать обычные и строковые кортежи необходимо просто проверить наличие индекса в массиве индексов равного положению кортежа в файле.

Добавление элемента не представляет из себя ничего интересного, узел просто добавляется в конец файла, а его идентификатор – в конец массива идентификаторов. Задачи фрагментации памяти делегирована удалению.

```c
enum crud_operation_status insert_new_cortege(FILE *file, struct cortege *cortege, size_t full_cortege_size, uint64_t *cortege_pos) {
    fseek( stream: file, off: 0, whence: SEEK_END);
    *cortege_pos = ftell( stream: file);//количество байт от начала файла
    int fd = fileno( stream: file);//определяет дескриптор файла открытого потока данных
    enum crud_operation_status status = ftruncate(fd, length: ftell( stream: file) + full_cortege_size);
    status = status | write_cortege(file, cortege, cortege_size: full_cortege_size - sizeof(union cortege_header));
    return  status;
}
```

Обновление элемента тоже происходит довольно тривиально – просто обновляем поле внутри узла. Однако в случае обновления строкового поля могут понадобиться новые строковые узлы, то есть необходимо расширить двусвязный список.

```c
enum crud_operation_status update_string_cortege(FILE *file, uint64_t offset, char *new_string, uint64_t size) {
    struct cortege *current_cortege;
    //Сдвигаемся на позицию обновляемого кортежа
    fseek( stream: file, off: offset, whence: SEEK_SET);
    //Читаем кортеж, который необходимо обновить
    read_cortege( cortege: &current_cortege, file, pattern_size: size);
    int64_t len = strlen( s: new_string);
    uint64_t old_offset = offset;
    //пока существует следующий кортеж (current_cortege->header.next) и есть данные для обновления (len > 0)
    do {
        //Очистка старых данных
        free_cortege( cortege: current_cortege);
        offset = old_offset;
        fseek( stream: file, off: offset, whence: SEEK_SET);
        read_cortege( cortege: &current_cortege, file, pattern_size: size);
        fseek( stream: file, off: offset, whence: SEEK_SET);
        current_cortege->data = (uint64_t * )(new_string);
        //Перемещение указателя new_string на следующий блок данных размером size.
        new_string = new_string + size;
        write_cortege(file, cortege: current_cortege, cortege_size: size);
        old_offset = current_cortege->header.next;
        len -= size;
    } while (current_cortege->header.next && len > 0);
    uint64_t fpos;

    //Если остались еще данные
    if (len > 0) {
        insert_string_cortege(file, string: new_string, cortege_size: size, par_pos: offset, str_pos: &fpos);
        current_cortege->header.next = fpos;
        fseek( stream: file, off: offset, whence: SEEK_SET);
        write_cortege(file, cortege: current_cortege, cortege_size: size);
        free( ptr: current_cortege);
    }

    return CRUD_OK;
}
```

Поиск элемента также является довольно интуитивной операцией: пробегаемся по массиву идентификаторов, при необходимости заглядывая внутрь каждого узла, и ищем совпадение по нужному полю.

Самой интересной и сложной операцией является **удаление**. Так как наши кортежи одинакового размера, то при удалении нам необходимо взять крайний элемент и переместить его на место удаляемого.

```
enum crud_operation_status swap_last_cortege_to(FILE *file, uint64_t offset, uint64_t size) {
    uint64_t full_size = size + sizeof(union cortege_header);
    fseek( stream: file,  off: 0,  whence: SEEK_END);
    fseek( stream: file,  off: -(int64_t) full_size,  whence: SEEK_END);
    enum crud_operation_status status = swap_cortege_to(file,  pos_to: offset,  pos_from: ftell( stream: file),  cortege_size: full_size);
    fseek( stream: file,  off: -(int64_t) full_size,  whence: SEEK_END);
    status = status | ftruncate( fd: fileno( stream: file),  length: ftell( stream: file));
    return status;
}
```

Но при этом возникает необходимость изменения ссылок на кортежи в массиве индексов. Для обычных кортежей требуется просто поменять индекс в массиве. Для строковых же был применен иной подход. Так как строковый кортеж представляет из себя двусвязный список, то у первого элемента этого списка в качестве указателя на предыдущий элемент мы указали ссылку на обычный кортеж, к которому принадлежит строчный. В момент удаления нам требуется просто поменять данную ссылку.

Так же так как основной структурой данных является документное дерево, то при удалении элемента было необходимо удалять его дочерние элементы. Для этого был реализован алгоритм поиска в глубину:

```
static enum crud_operation_status remove_recursive_cortege_with_values
        (FILE *file, uint64_t id, uint32_t *types, size_t pattern_size) {

    uint64_t size = get_cortege_size(pattern_size);
    uint64_t offset = remove_from_id_array(file, id);
    if (offset == NULL_VALUE) return CRUD_INVALID;
    struct uint64_list *children = get_children_by_id(file, id);
    for (struct uint64_list *j = children; j != NULL; j = j->next) {
        remove_recursive_cortege_with_values(file,  id: j->value, types, pattern_size);
    }
    struct cortege *current_cortege;
    fseek( stream: file,  off: (int32_t) offset,  whence: SEEK_SET);
    read_cortege( cortege: &current_cortege, file, pattern_size);
    for (size_t i = 0; i < pattern_size; i++) {
        if (types[i] == STRING) {
            //Remove string from file
            union cortege_header cur_header;
            uint64_t temp_offset = current_cortege->data[i];
            while (temp_offset != NULL_VALUE) {
                fseek( stream: file,  off: current_cortege->data[i],  whence: SEEK_SET);
                read_from_file( buffer: &cur_header, file,  size: sizeof(union cortege_header));
                temp_offset = cur_header.next;
                swap_last_cortege_to(file,  offset: current_cortege->data[i], size);
                current_cortege->data[i] = temp_offset;
            }
        }

    }
    free_uint64_list( result: children);
    free_cortege( cortege: current_cortege);
    return swap_last_cortege_to(file, offset, size);
}
```

Также необходимо было расширять массив индексов, для этого мы использовали тот же функционал, но перемещали элемент из начала в конец, тем самым расширяя массив.

```
enum crud_operation_status append_to_id_array(FILE *file, uint64_t offset) {
    enum crud_operation_status status = CRUD_OK;
    struct tree_header *header = malloc( size: sizeof(struct tree_header));
    size_t array_pos;
    read_tree_header_no_id(header, file,  fpos: &array_pos);
    uint64_t real_id_array_size = get_id_array_size(header->subheader->pattern_size, header->subheader->cur_id);
    //Если недостаточно места
    if (!((header->subheader->cur_id + 1) % real_id_array_size)){
        uint64_t from = ftell( stream: file) + real_id_array_size * sizeof(uint64_t);
        fseek( stream: file,  off: 0,  whence: SEEK_END);
        uint64_t cur_end = ftell( stream: file);

        //Освобождаем место под новый кортеж
        status = status | ftruncate( fd: fileno( stream: file),  length: cur_end + get_cortege_size(header->subheader->pattern_size) + sizeof(union cortege_header));
        //Заполняем
        swap_cortege_to(file,  pos_to: cur_end,  pos_from: from,
                        cortege_size: get_cortege_size(header->subheader->pattern_size) + sizeof(union cortege_header));
    }
    write_id_value(file,  fpos: array_pos, offset,  index: header->subheader->cur_id++);
    write_tree_header_no_id(file, header);
    free_tree_header_no_id(header);
    return status;
```

Результаты:

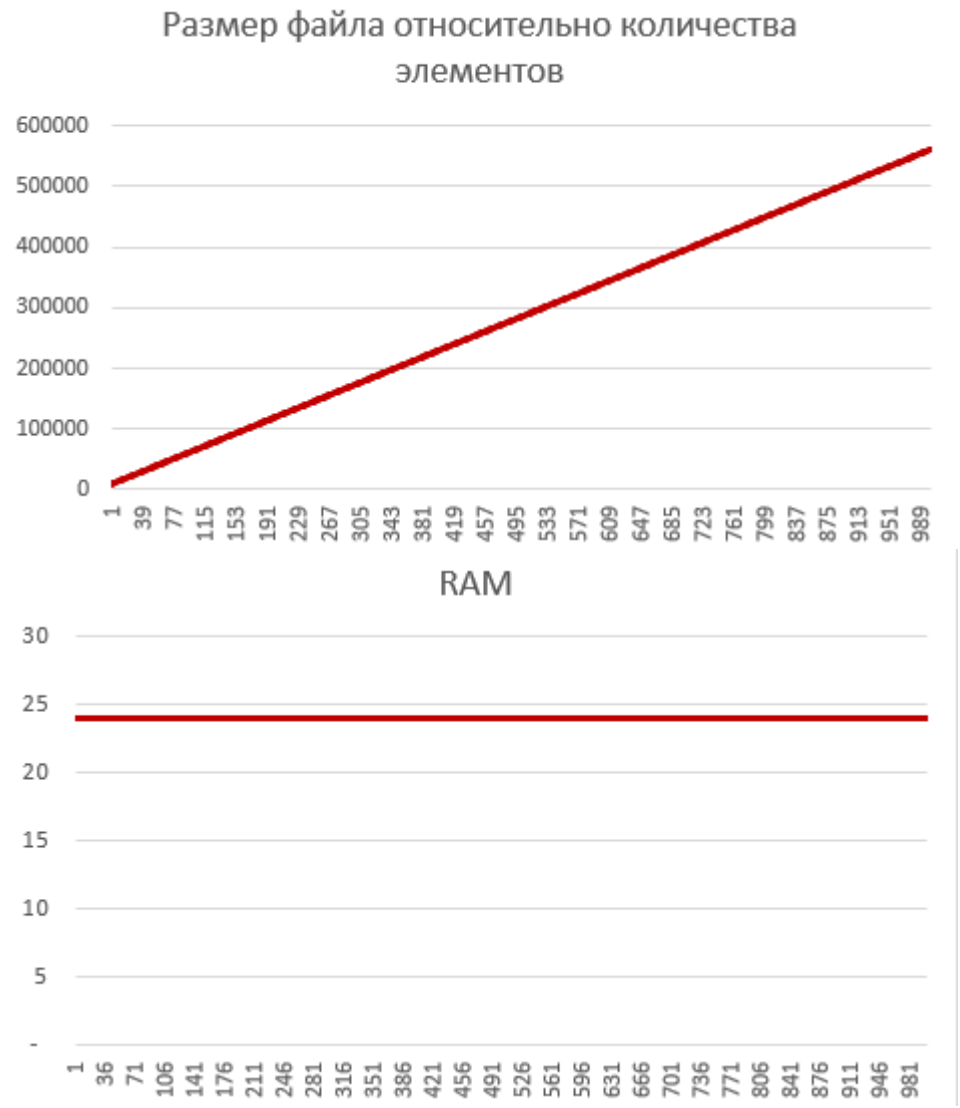
Для выполнения задач было:

- Созданы соответствующие структуры данных
- Описаны базовые операции с данными
- Создан публичный интерфейс для работы с операциями
- Созданы тестовые сценарии для демонстрации работоспособности и эффективности решения

Алгоритмическая сложность:
Вставка – O(1)
Обновление – O(1)
Удаление – O(m), где m – количество зависимых элементов
Поиск по id – O(1)
Поиск по родителю – O(1)
Поиск по полю - O(m), где m - количество представленных элементов данных выбираемого вида

Изначально было 1000 элементов, каждый зависит от каждого. Чем меньше зависимых элементов, тем выше скорость удаления.

Чем больше элементов, тем ниже скорость поиска.



Выводы:

Тесты показали, что предложенная структура удовлетворяет требованиям производительности. Такие показатели достигаются благодаря идеи хранения массива индексов – отступов. Это позволяет за константу находить элемент по id. Также добавление занимает константное время,

так как элемент добавляется в конец. Идея с разделением кортежей на строковые и обычные даёт потенциальную возможность хранить в файле строки любого размера. Есть один минус данной реализации – при перемещении кортежей необходимо обновлять индексы.